Augur: Dynamic Taint Analysis for Asynchronous JavaScript

Mark W. Aldrich
Tufts University
Medford, MA, USA
mark.aldrich@tufts.edu

Alexi Turcotte
Northeastern University
Boston, MA, USA
turcotte.al@northeastern.edu

Matthew Blanco
Northeastern University
Boston, MA, USA
blanco.m@northeastern.edu

Frank Tip
Northeastern University
Boston, MA, USA
f.tip@northeastern.edu

ABSTRACT

Dynamic taint analysis (DTA) is a popular approach to help protect JavaScript applications against injection vulnerabilities. In 2016, the ECMAScript 7 JavaScript language standard introduced many language features that most existing DTA tools for JavaScript do not support, e.g., the `async`/`await` keywords for asynchronous programming. We present Augur, a high-performance dynamic taint analysis for ES7 JavaScript that leverages VM-*supported* instrumentation. Integrating directly with a public, stable instrumentation API gives Augur the ability to run with high performance inside the VM and remain resilient to language revisions. We extend the abstract-machine approach to DTA to handle asynchronous function calls. In addition to providing the classic DTA use case of injection vulnerability detection, Augur is highly configurable to support any type of taint analysis, making it useful outside of the security domain. We evaluated Augur on a set of 20 benchmarks, and observed a median runtime overhead of only 1.77×, a median performance improvement of 298% compared to the previous state-of-the-art.

KEYWORDS

dynamic program analysis, taint analysis, information flow analysis, security vulnerabilities, JavaScript

ACM Reference Format:
Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. 2022. Augur: Dynamic Taint Analysis for Asynchronous JavaScript. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3559522

1 INTRODUCTION

JavaScript dominates the online world [8], powering everything from client-side web apps to server back-ends. Web applications are frequently the target of cyberattacks, and subtle bugs in JavaScript can lead to many dangerous vulnerabilities, including format-string attacks, SQL injection, cross-site scripting, command- and shell-code injection, and directory traversal.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3559522

Dynamic taint analysis (DTA) has a long history of detecting the above vulnerabilities. It is particularly effective in finding vulnerabilities in dynamic languages like JavaScript, as their dynamic behavior is difficult to analyze using static program analysis. Unfortunately, many existing DTA tools for JavaScript [1, 3–5, 10] do not support the JavaScript language features introduced in ECMAScript 7, and have significant overhead on the versions of JavaScript that they do support, ranging from 3.38× [6] to 1,680× [1].

In this paper, we present Augur, a dynamic taint analysis tool for JavaScript. The technique underpinning Augur is one of VM-supported instrumentation; Augur is implemented in the NodeProf [9] framework for GraalVM [11], which exposes a stable instrumentation API upon which to build a dynamic analysis. This scheme allows Augur to achieve low overhead (median 1.77× over 20 benchmark applications), and avoiding the need to modify the VM itself significantly reduces engineering effort and increases the likelihood of adoption. Augur builds upon the abstract-machine approach introduced by Ichnaea [5], expanding upon it to handle the new asynchronous features of JavaScript ES7.

2 BACKGROUND & MOTIVATION

DTA is a program analysis technique to track the flow of sensitive information and ensure it does not reach an untrusted operation. Confidential or sensitive information is considered *tainted* when it comes from a taint *source*, and leaked when it enters a taint *sink*. While the technique was originally intended to detect privacy leaks, it has many additional applications such as detecting security vulnerabilities. For example, SQL injection attacks can be detected using DTA by marking user input as taint sources, and SQL commands as sinks. If user input flows into an SQL command without sanitization, even indirectly, a user can potentially exploit this vulnerability by writing raw SQL code.

DTA is particularly effective for JavaScript, as the dynamic nature of the language can make it difficult to determine these vulnerabilities statically. However, JavaScript's design presents some unique challenges for DTA. Modern JavaScript is Just-In-Time (JIT) compiled and optimized. This greatly improves its performance, but makes it infeasible to take advantage of lower-level x86 binary DTAs [2]. This necessitates a higher-level instrumentation mechanism. However, JavaScript code utilizes an extensive standard library implemented in native code, meaning any DTA implementation needs to precisely track data flow in both JavaScript and native code. Existing JavaScript DTAs implement the instrumentation in

```
$ node ~/augur/ts/runner/cli.js --projectDir . \
       --projectName weather --outputDir . --main weather.js

Tainted value flowed into sink:
{"type":"variable","name":"z","location":{"fileName":"weather.js"}}
```

Figure 1: AUGUR analyzing an application via the terminal. Here, AUGUR is running on the `weather` project (in the current directory, `.`), with `weather.js` as the main file. In this example, AUGUR reports a taint flow in `weather.js`.

one of two ways: either via program rewriting [1, 3–5], or VM modification [6, 10], and most (besides TruffleTaint [6]) do not support the asynchronous features introduced in JavaScript ES6 and ES7.

Introduced in ES6, promises are a popular method of implementing asynchronous computations in JavaScript. A promise can be thought of as an object which wraps an asynchronous computation; reaction *callbacks* can be registered on promises that are invoked when the underlying asynchronous computation is completed, either with the return value if execution was successful, or with an error if a problem occurred. The ES7 version of JavaScript has introduced the `async/await` feature which is not supported by most DTA tools. At a high level, `async/await` is syntactic sugar for promises: by marking functions as `async` and `await`-ing calls to such `async` functions, programmers can write asynchronous code with linear-looking control flow. AUGUR leverages the analysis technique used in Ichnaea with a modern implementation to support the latest version of JavaScript, and offers improved performance through to VM-supported instrumentation.

3 AUGUR OVERVIEW

AUGUR is a dynamic taint analysis platform for JavaScript. It is implemented in the NodeProf [9] instrumentation framework for GraalVM [11]. It supports the latest version of JavaScript, including `async/await`, native Promises, classes, as well as new syntax: `let`, `const`, `for/of`, and arrow functions. Taint can be tracked through native functions, with built-in models for many popular functions. AUGUR is equipped with a default specification that searches for common injection vulnerabilities in JavaScript: file contents and process arguments are marked as sources, and `exec` and `eval` (which can execute arbitrary strings as code) are marked as sinks (this spec is fully customizable). Detailed instructions on how to run AUGUR and how to configure a taint specification are available in the open-source repository[1]; as a simple example of how to run AUGUR on the command line, consider Figure 1.

4 AUGUR'S TECHNIQUE & IMPLEMENTATION

AUGUR follows in the footsteps of Ichnaea [5], where code is first instrumented to produce instructions for a *stack machine*. The stack machine is then responsible for finding the information flows.

[1]See https://github.com/nuprl/augur.

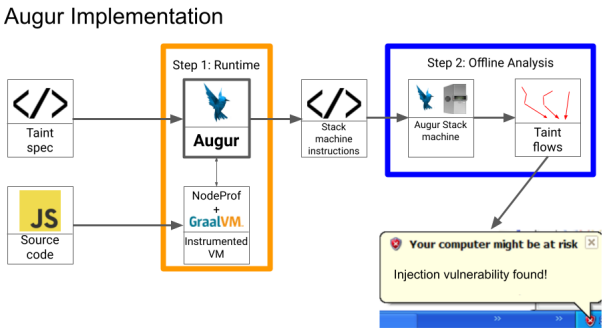

Figure 2: AUGUR overview.

Run Time Analysis. A JavaScript program is loaded into GraalVM for execution. As it runs, GraalVM, through the NodeProf framework, informs AUGUR of every operation happening in the program—e.g., variable assignments, function calls, etc.—via hooks. AUGUR's run time analysis takes this information and produces instructions for AUGUR's stack machine that describe how information flows while executing the operation. These instructions represent an execution trace which AUGUR can later analyze for data flow.

Offline Analysis. Here, AUGUR runs the stack machine instructions in the execution trace w.r.t. a *taint specification*, describing which program locations are *sources* of taint and which are *sinks*.

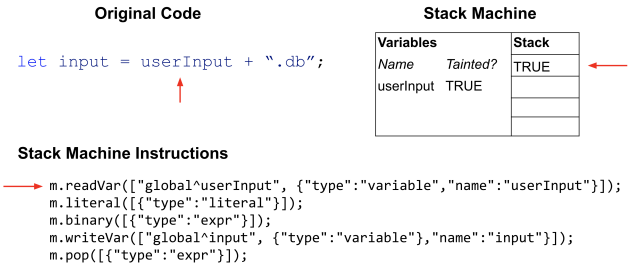

Figure 3: Stack machine instructions example.

Figure 3 displays the stack machine instructions and state of AUGUR when instrumenting the code snippet under "Original Code", configured with `userInput` as a taint source and `input` as a taint sink. When the first line of the stack machine instructions is executed, AUGUR pushes `true` onto the stack to indicate a tainted value (note the red arrows in the figure). The next instruction causes `false` to be pushed onto the stack, indicating an untainted literal value. Next, a binary operation corresponding to the addition in the original code causes AUGUR to merge the two values at the top of the stack: `true` and `false` are popped off the stack and combined into `true` with boolean `or`, which is pushed onto the stack (describing the taintedness of the combined string literal). In the next instruction, the stack machine assigns the taint value at the top of the stack to the `input` variable and reports any flows that occurred. Since the value at the top of the stack is `true`, AUGUR has

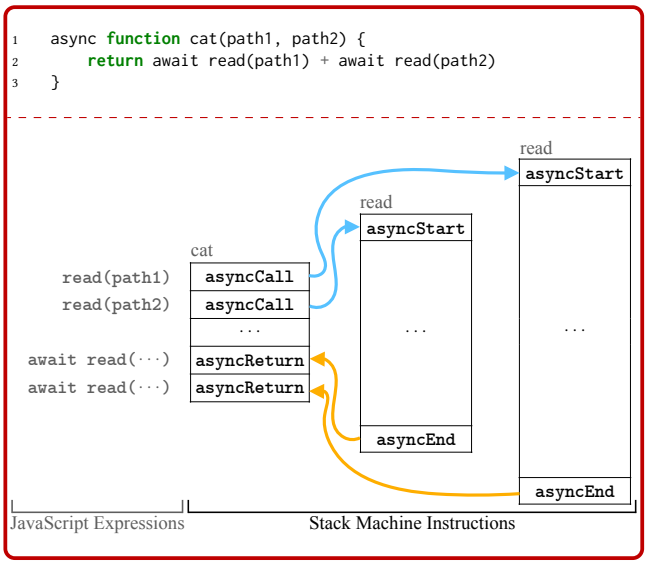

Figure 4: How asynchronous operations in JavaScript are translated into stack machine instructions with separate stacks.

now determined that the variable `input` is tainted, and since `input` is defined as a sink, the flow will be reported to the user.

This is the technique described in Ichnaea [5], and in the following subsections we describe extensions for handling ES7 features.

4.1 Async/Await Support

To support dynamically pausing and resuming asynchronous function calls, AUGUR is implemented with a *taint tree* rather than a taint stack. In a sense, there are separate taint stacks for each asynchronous function call, and AUGUR links up calls to asynchronous functions, their return values, and their callers with a unique ID associated with each execution of an asynchronous function. Halting execution of an `async` function will save the current taint stack and associated with the unique ID of that function execution; when execution later resumes, the saved stack is retrieved and instrumentation resumes with the appropriate taint stack in place. The taintedness of return values from `async` functions is also explicitly saved alongside the ID of the call so that once the *caller* of that function resumes execution, the appropriate taint can be retrieved.

As an example, consider the program in Figure 4. The asynchronous function `cat` calls the asynchronous `read` function twice. Each function call has its own independent taint stack, where the taint of arguments and return values are linked.

4.2 Instrumenting Promises

Consider the following code snippet:

```
1  new Promise((resolve, reject) => resolve(x))
2      .then(y => eval(y));
```

A promise is created on line 1, which resolves (asynchronously) with the value `x`. This value flows into the callback registered as a reaction on the promise on line 2, where `y` will take on the value that the promise resolved with. AUGUR is equipped with models for native `Promise` functions that track flow between promises and their reactions, with unique IDs for each promise and context switching between taint stacks associated with each promise execution.

Table 1: Results of running AUGUR on benchmarks from the Ichnaea paper [5]. The first row of the table reads: the `coco-utils` benchmark ran in 67.05ms without instrumentation, and in 117.73ms with AUGUR instrumentation. The overhead of AUGUR's run time analysis is 1.76×, and Ichnaea reported an overhead factor of 6.39×. AUGUR generated 1,060 stack machine instructions. (*) `mongoosify-eval` and `pidusage-exec` do not run on modern Node.js and GraalVM.

Benchmark	Runtime (ms)		Overhead (factor)		# generated instructions
	Original	AUGUR	AUGUR	Ichnaea	
cocos-utils	67.05	117.73	1.76	6.39	1,060
chook-growl-reporter-exec	72.34	178.34	2.47	6.7	1,255
fish-exec	66.33	102.05	1.54	3.17	492
git2json-exec	61.72	149.92	2.43	4.83	1,363
gm-attack	79.99	142.68	1.78	13.62	3,115
growl-exec	62.76	117.16	1.87	4.57	799
libnotify-exec	61.45	100.89	1.64	3.34	561
m-log-eval	18.79	87.80	4.67	12.48	11,112
mixin-pro-eval	16.59	18.05	1.09	5.14	392
modulify-eval	9.16	582.47	63.58	29.42	15,563
mongo-parse-eval	17.04	17.17	1.01	7.28	598
mongoosemask-eval	20.54	185.56	9.04	21.04	16,341
mongoosify-eval*	–	–	–	–	–
node-os-utils	73.68	130.10	1.77	8.19	1,292
node-wos	70.45	126.42	1.79	4.65	951
office-converter	75.03	113.87	1.52	3.53	512
os-uptime	75.20	118.19	1.57	3.4	441
osenv	15.95	17.46	1.09	4.66	752
pidusage-exec*	–	–	–	–	–
pomelo-monitor	66.26	115.15	1.74	4.13	699
system-locale	48.83	125.74	2.58	3.54	554
systeminformation	11.89	44.38	3.73	25.24	21,833

4.3 Generalized Taint Analysis

While dynamic taint analysis is most popularly used for detecting injection vulnerabilities, the technique is capable of determining any or all data flows within a program. Many research DTAs focus on supporting injection vulnerabilities over other types of flows, and AUGUR's technique and implementation was designed to be generalized and supports implementing *any* type of taint analysis.

A taint analysis in AUGUR can be described as a tuple $(V, F, L, \cup) \rightarrow \bar{F}$, where V is the type used to represent a taint value, F is the type used to represent a taint flow, L is a labelling function, and $\cup$ is a join function. The output of a taint analysis is a list of flows of type F. For detecting injection vulnerabilities, only a simple taint analysis is required: (boolean, *loc*, *isSource*, $\vee$).

4.4 VM-Supported Instrumentation

AUGUR takes a novel *VM-supported* approach to instrumentation. AUGUR is built in the NodeProf framework, an officially supported JavaScript instrumentation API for Oracle's GraalVM. By hooking into NodeProf within the VM, AUGUR can fully observe program execution without source code or VM modification. NodeProf then provides AUGUR with lower-level information related to data flow, making the transition to stack instructions more simple.

5 EVALUATION

We ran Augur on the benchmarks used in the Ichnaea paper [5]. This benchmark suite is comprised of 22 real JavaScript applications known to present 2 common injection vulnerabilities in JavaScript: `eval` (evaluates arbitrary code) and `exec` (executes arbitrary shell commands). Note that two of the benchmark applications no longer execute, even on base Node.js, due to changes in the JavaScript standard library, so we focus on the 20 that do. The results of this experiment, alongside a comparison with Ichnaea overhead, can be found in Table 1. Overall, we found that Augur achieves a low median run time overhead of 1.77× on the 20 benchmark applications that execute, and a median overhead of 3.35× when executing generated stack machine instructions alongside program execution. Augur outperforms Ichnaea on 17 of the 20 benchmarks. We investigated the cases where Ichnaea outperformed Augur, and found no systematic reason; one possible reason is due to engineering differences, as Ichnaea is implemented using source code rewriting (running on V8), whereas Augur is implemented in the NodeProf framework for GraalVM. In both Augur and Ichnaea, the time to execute the stack machine instructions is negligible.

Additionally, we conducted extensive testing to validate the correctness of our taint tracking through `async/await`, Promises, and the interaction between the two. We found no instances where taint was lost when switching contexts.

6 LIMITATIONS

Augur may not be able to analyze JavaScript code out of the box due to native functions. Although we have implemented 20 models for common native functions, JavaScript continues to enrich its standard library with each release. That said, only 76 different native models were invoked during our evaluation and no issues were observed, suggesting that the default model covers many unimplemented native models. Separately, Augur is built in the NodeProf dynamic analysis framework for Truffle and GraalVM. GraalVM fully supports the JavaScript language, but falls short of integrating directly with web browsers; to achieve that, mock browser environments (e.g., `jsdom`) should be used.

7 RELATED WORK

While there is a wealth of work on taint analysis in general, *dynamic* taint analysis is most closely related to Augur. In this section, we outline dynamic taint analysis tools for JavaScript specifically. Dynamic taint analysis can be broadly categorized as either rewriting the program, or by modifying a virtual machine.

Program Rewriting. Ichnaea [5] relies on Jalangi [7] to perform program rewriting; Ichnaea's stack machine approach is similar to the approach employed in Augur, but it is limited in that (1) Jalangi is no longer supported, and does not work on modern asynchronous JavaScript, and (2) Ichnaea has significantly more overhead than Augur (Ichnaea median overhead is 5.92×). Other work includes a Virtual Values approach [4] wherein taint flow semantics are declared for primitive JavaScript operations, and an approach where information flow controllers are explicitly inlined [1].

VM Modification. Recent work by Kreindl et al. [6] propose *TruffleTaint*, a platform-agnostic taint analysis for Truffle [11], the framework for implementing language runtimes in GraalVM [11]. While Augur is implemented in the NodeProf framework [9] and thus runs on unmodified GraalVM, *TruffleTaint* runs on a fork of GraalVM (reported overhead 3.32×-7.30×). JSFlow [3] enhances the JavaScript interpreter to track fine-grained information flow (overhead of 1,680× reported in other work [1]), and the XSS Prevention approach taken by Vogt et al. [10] relies on a modified browser.

The VM-supported approach outlined in this paper yields greater performance than program rewriting techniques [1, 3–5] as Augur does not modify source code and is more portable than techniques which require VM modifications [6, 10].

8 CONCLUSION

Dynamic taint analysis is an important technique to help secure JavaScript applications against security vulnerabilities. Many existing DTA techniques and tools for JavaScript have not been updated to handle the ES7 version of JavaScript, which introduced the `async/await` feature to the language. This paper presented Augur, the most performant DTA tool for JavaScript that supports JavaScript ES7; thanks to VM-supported instrumentation, Augur achieves a low median run time overhead of 1.77× on 20 benchmark applications, and a median overhead of 3.35× when executing generated stack machine instructions alongside program execution.

ACKNOWLEDGMENTS

This research was supported by National Science Foundation grants CCF-1715153 and CCF-1907727, and by a gift from Oracle Labs.

REFERENCES

[1] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 629–643.
[2] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 196–206.
[3] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.
[4] Prakasam Kannan, Thomas Austin, Mark Stamp, Tim Disney, and Cormac Flanagan. 2016. Virtual values for taint and information flow analysis. (2016).
[5] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020
[6] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2020. Multi-language dynamic taint analysis in a polyglot virtual machine. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. 15–29.
[7] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.
[8] Stack Overflow. 2021. 2021 Developer Survey. See https://insights.stackoverflow.com/survey/2021.
[9] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient dynamic analysis for Node. js. In *Proceedings of the 27th International Conference on Compiler Construction*. 196–206.
[10] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, Vol. 2007. 12.
[11] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204.